ISI Research Report
ISI/RR-89-223
July 1989

4

University of Southern California

Scott Carter
David Mizell

ISI's SDI Architecture Simulator: An Experiment in Adding New Software to the System

INFORMATION SCIENCES INSTITUTE ISI
213/822-1511
4676 Admiralty Way/Marina del Rey/California 90292-6695

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT This document is approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) ISI/RR-89-223	5. MONITORING ORGANIZATION REPORT NUMBER(S) -------
6a. NAME OF PERFORMING ORGANIZATION USC/Information Sciences Institute 6b. OFFICE SYMBOL *(If applicable)*	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research
6c. ADDRESS *(City, State, and ZIP Code)* 4676 Admiralty Way Marina del Rey, CA 90292-6695	7b. ADDRESS *(City, State, and ZIP Code)* 800 N. Quincy Street Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SDIO 8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N0014-87-K-0022
8c. ADDRESS *(City, State, and ZIP Code)* Strategic Defense Initiative Orgainzation Office of the Secretary of Defense The Pentagon, Washington, DC 20301	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
-------	-------	-------	-------

11. TITLE *(Include Security Classification)* (Unclassified)
ISI's SDI Architecture Simulator: An Experiment in Adding New Software to the System

12. PERSONAL AUTHOR(S)
Carter, Scott; Mizell, David

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Research Report	FROM ______ TO ______	1989, July	

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	discrete-event simulation, modular software design
09	02		object oriented programming; SDI architecture;
			software engineering

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

The design of ISI's SDI architecture simulator was intended to minimize the software development necessary to add new simulation models to the system or to refine the detail of existing ones. The key software design approach used to accomplish this goal was the modeling of each simulated defense system component by a software object called a "technology module." Each technology module provided a carefully defined abstract interface between the component model and the rest of the simulation system, particularly the simulation models of battle managers. This report documents our first test of the validity of this software design approach. A new technology module modeling a "kinetic kill vehicle" (KKV) was added to the simulator. Although this technology module had an impact on several parts of the simulation system in the form of new data structures and functions that had to be created, the integration of the new module was accomplished without the necessity of replacing any existing code.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified	
22a. NAME OF RESPONSIBLE INDIVIDUAL Victor Brown Sheila Coyazo	22b. TELEPHONE *(Include Area Code)* 213/822-1511	22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted.
All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

ISI Research Report
ISI/RR-89-223
July 1989

University of Southern California

Scott Carter
David Mizell

ISI's SDI Architecture Simulator: An Experiment in Adding New Software to the System

INFORMATION SCIENCES INSTITUTE
213/822-1511
4676 Admiralty Way/Marina del Rey/California 90292-6695

This research is supported by the Strategic Defense Initiative Office under Office of Naval Research Contract No. N00014-87-K-0022. Views and conclusions contained in this report are the authors' and should not be interpreted as representing the official opinion or policy of the Strategic Defense Initiative Office, the Office of Naval Research, or the U. S. Government. No official Government endorsement should be inferred.

1. Introduction

Throughout 1987, the ISI parallel and distributed computing research group designed and implemented a prototype sequential simulator of SDI architectures. A discussion of our design approach and a detailed description of the resulting design are provided in [1].

The system is designed to be used for high-level simulations of candidate defense system architectures, where the user is willing to give up full fidelity of the simulation models for the sake of doing full, "end-to-end" simulations of a candidate architecture, complete with simulation models of all major system components, in order to assess how well the battle management system makes these components work together. We expected, however, that users would want to increase the amount of detail and realism in the simulation models repeatedly, as they narrowed the field of candidate architectures to a subset considered worthy of closer examination. Accordingly, we sought to structure the simulation software design in a way that minimized the effort necessary for programmers responsible for future system enhancements to add new simulation models to the system or to add more detail to existing models.

This report documents our first experiment aimed at testing the hypothesis that our software structure is indeed conducive to the addition/refinement of simulation models. This experiment involved the design, implementation, and incorporation into the simulation system of a simulation model of a "kinetic kill vehicle" (KKV) weapon -- a small rocket designed to be launched from an orbiting defense platform at an enemy missile at such a velocity that the combined kinetic energies of the rocket and its target would destroy the target. We had not modeled this type of weapon in our simulator before. Our simple debugging simulation models included only an abstract, idealized representation of a laser, which projected energy toward the target at the speed of light. One of the main reasons we chose to implement the KKV technology module was that we expected this implementation to be a fairly strenuous test of the software structure's ability to minimize the effort necessary to add new simulation models. It required changes in several places in the simulation system, including the addition of new types of simulation events, a new type of physical object, a new type of trajectory, and a new platform "capability" -- which is a data structure that specifies a component that can be attached to a platform (see [1]). Also, it required that battle management abstractions (BMAs) be designed to take into account the significant delay between the firing of a weapon and its hitting or missing the target. A BMA might not be able to wait to observe the result of a KKV launch; the constraints may well be such that the attack launch window (assuming the target must be attacked in boost phase) on a given target ends before the first interception would occur. This suggests that fundamentally different BMA strategies might be required for KKVs as compared to speed-of-light weapons.

Section 2 of this report describes the software design approach we used to try to cleanly separate the simulation models of environment and system components from the model of the candidate architecture being simulated. Section 3 discusses the general implications of this software modularity approach on subsequent attempts to increase the level of detail in simulation models. Section 4 describes the design of the new KKV technology module. Section 5 describes the results obtained from the first simulations that we ran using the KKV technology module and the BMAs that we modified to use it. Section 6 presents our conclusions from this experiment and our suggestions for future work on this topic. The Appendix provides a detailed summary of the software modules that were developed or revised in the course of implementing the KKV technology module.

2. Overview of the "Technology Module" Design Approach

We used the term "technology modules" in [1] to refer to software modules in the simulation system that model components of a defense system platform. They include sensors, weapons, communications capabilities, and propulsion capabilities of any platform. Technology modules are centrally significant to our approach to the design of the simulation system software, in that they constitute the interface between a BMA and the simulated environment. The only way the state of a given BMA can be affected by events external to its platform is via technology modules. For example, a BMA's state would be changed by the arrival of a message from another platform via its communication technology module. With the exception of the BMA initialization events at the beginning of the simulation run and delay events (and the former are artifacts of the implementation that are not intended to model real system behavior), a platform's BMA will be scheduled for execution in our simulation system only if one of its technology modules posts an event in the event queue to which that BMA is programmed to respond. Likewise, the only way that a BMA is able to affect the "outside world" -- that is, to cause events capable of changing the state of the system external to the BMA's platform -- is by executing calls to procedures associated with particular technology modules. For example, the BMA on a particular platform can change the state of the external environment by launching one or its weapons or by sending messages to one or more other platforms, either of which is accomplished by calling a function within the appropriate technology module.

Thus, it is the representation of a defense platform's various components as individual software modules in our simulator that enabled us to establish a clean separation between the simulation model of the external physical environment and that of the candidate defense architecture. Also, it provided us a mechanism for incorporating abstract representations of executing battle management systems within a discrete-event simulator. Finally, this approach has the additional payoff of allowing these components to be selected by users for different candidate defense architectures to be simulated. This reduces potential software development effort by allowing these modules to be reused. More importantly, it contributes to the evaluation of different candidate defense architectures under the same set of underlying assumptions about the way sensors, weapons, communication systems and other defense platform components work.

3. Design Issues in the Incremental Refinement of Simulation Models

Almost all simulation models are abstractions of the actual entity being modeled. The technology modules that we inserted into our prototype simulation system are especially simplified abstractions. For example, our sensor module always provides error-free data to its BMA. That is, the data that the sensor hands to the BMA on its platform is "ground truth" information about the position and velocity of all targets within its field of view. There is another technology module that models a packet radio system used for interplatform communication. While this module does take into account a possibility of a message failing to arrive at its destination (by using a decreasing exponential function of distance between sender and receiver), it in no way takes into account the possibility of communication being interfered with as a result of an external phenomenon such as a nuclear explosion or enemy jamming.

Simulations of this sort will always be abstractions to a considerable degree. Systems as complex as the defense architectures being modeled here have a hierarchy of increasing levels of detail that is so broad and so deep that one cannot imagine fully realistic simulation models ever being completely programmed, much less being executed on any feasible computer system. This is not by any means a reason to abandon the futher development of this type of simulation system, however. It simply implies that such a system cannot be used for the detailed debugging of any hardware or software component. Such use was never our intention. Our original premise was that it would both be technically feasible and worthwhile for SDIO to develop a simulation model for defense architectures that did not contain fully detailed models of any system component, but did contain models of *all* system components, so that users could conduct "end-to-end" simulation experiments designed to examine at a high level how the battle management system made all these components work together.

We certainly expect users of this type of defense architecture simulation system to require much greater levels of detail in their technology modules than is present in our debugging prototypes. Indeed, we expect that the level of detail in the technology modules of the simulator will repeatedly be increased throughout its lifetime. On the other hand, we also expect that the technology modules will always be abstract simplifications of the actual physical components. The key implication is that whenever the level of detail in a technology module is changed, a considerable amount of careful design effort must be applied to the interface between the newly refined technology module and the BMAs that use it. The central interface design issue is that of how much the BMA is to "know" about the technology module. If the programmer of the BMA does not have adequate information about the behavioral characteristics of the system component, as represented by its technology module, then he or she cannot design a BMA that uses the system component as effectively as possible. On the other hand, if BMA programmers are fully apprised of the details of the technology module implementation, then they may tend to write "pathological" BMAs that attempt to exploit idiosyncrasies in the implementation, rather than only basing their BMA design on the subset of the system component's characteristics that the technology module is designed to faithfully represent. At best, the BMA programmer may become entangled in a mass of detail that is largely irrelevant to the task of determining whether or not the battle management system is able to make defense system components work together effectively.

The approach we have devised for controlling the level of abstraction in the interface between a BMA and a technology module is to associate every technology module with a set of procedures called "BMA helpers." These are procedures that the BMA can call in order to obtain information that helps the BMA use the system component effectively. For example, the technology module that represents a particular type of weapon might be augmented by a BMA helper function which, when given the current locations and velocitites of both a weapon platform and a prospective target, returns the probability of kill of a weapon launched from that platform at that target. Another BMA helper returns the optimal launch time for a weapon platform against a given target -- that is, the launching time that would give that platform the highest probablity of kill against that target. Thus, BMA helper functions, as well as the technology module procedures themselves, perform computations that would naturally be part of the battle management system itself in either an actual system implementation or a more detailed simulation model. In both cases, these details are hidden from the BMA programmer so that he or she may concentrate on higher-level battle management system design issues.

4. Software Design of the KKV Technology Module

4.1 Physical Model of KKVs

The KKV technology module uses a highly simplified model of KKV behavior. KKVs are assumed to be launched from their carrier vehicle with a constant delta velocity in any direction (the current implementation includes no slewing time to point the launcher, though that is one of many possible extensions to the technology module that we considered). There is a certain minimum delay between successive launches from the same launcher. KKVs are aimed at a "target," which in this case is a data structure that allows the technology module (simulating a computational function on the platform) to extrapolate the future motion of the target missile.

The KKV follows a constant angular velocity path with constant dr/dt. While this assumption is not accurate in terms of even the most basic orbital mechanics, the error (a few percent in the track produced compared to a constant-energy track) is inconsequential in terms of the level of fidelity of the KKV model as a whole, and the simplification makes the simulation much less compute intensive.

The homing behavior of the KKV, including any track updates from its carrier platform, is abstracted into the single Pk calculation. The homing behavior is subject to the constraint that the interception must take place before the target missile stops boosting. Note that this and similar constraints could be easily changed entirely within the KKV technology module and its associated BMA helpers, without requiring any changes to the BMAs which use the technology module -- although a different BMA might perform better when the technology module constraints were changed. This is exactly the sort of question that the simulator is intended to be useful for addressing.

The probability of a successful interception in the model is a negative exponential function of the flight time of the KKV. Higher fidelity models might take into account the intercept angle, quality of the track data, etc.

4.2 Interfaces to BMAs and to the Simulator Kernel

A design goal of the simulator in general, and the KKV technology module in particular, has been to keep the BMA interfaces to similar system components reasonably consistent. In the case of the KKV, it was possible to make the main BMA interfaces to the KKV launcher and its BMA helpers identical to their laser weapon equivalents:

Laser	KKV launcher
shoot_at(target,platform_id)	shoot_kkv_at(target,platform_id)
pk(target,platform_id,time)	kkv_pk(target,platform_id,time)
opt_pk(target,platform_id)	kkv_opt_pk(target,platform_id)

kkv_pk(), which returns the expected pk for launching a KKV from a given platform at a target at a given time, and kkv_opt_pk(), which returns the optimal time to launch a KKV at a specific target from a given platform, are the two BMA helpers used by the existing BMA. There are also some "lower level" BMA helper functions, which are invoked by kkv_opt_pk(). These could be used by a BMA that was more intimately tied to the KKV's performance. Specifically, kkv_flight_time() and kkv_dfdl(), which returns the instantaneous derivative of flight time with respect to launch time, could be used to get an idea of the width of the KKV launch window for a near-optimal shot.

Two event-handling routines, execute_kkv_launch() and execute_kkv_intercept(), comprise the interface between the KKV technology module and the simulator kernel.

4.3 Sequence of Operations in the Technology Module

Figure 1 summarizes the sequence of operations that occur when a BMA uses the KKV technology module in the course of a simulation run. First, a BMA decides (probably based on information provided by the KKV BMA helpers) whether and when to launch a KKV at a target.

The BMA then calls the BMA interface procedure launch_kkv_at(). If the operational constraints of the KKV launcher are met (at least one KKV in the magazine/launcher bus, and sufficient time elapsed since the last launching), a KKV_LAUNCH event is posted for a short time into the future.

When the simulator evaluates the KKV_LAUNCH event, calling execute_kkv_launch(), a kkv (note lower case) software object is created (representing a physical KKV object in the simulation system). Based on current target parameters, the intercept time of the KKV with its target is calculated, and a KKV_INTERCEPT event is posted for that time.

When the simulator evaluates the KKV_INTERCEPT event, calling execute_kkv_intercept(), the Pk for the intercept is calculated based on the actual interception parameters (currently, only the KKV's flight time enters the calculation) and is compared to a random number. If the intercept is successful, a PDESTROY event is posted for the target missile.

At this point, the sequence of events is the same as for a missile destruction event caused by a laser weapon, as described in pp.17–18 of [1].

5. First Simulation Results

Complete source listings of the BMAs of a sensor platform and a weapon platform for a simple, laser-based strategic defense architecture are provided in [2]. These were modified in the course of this experiment to use KKVs instead of lasers. No other changes to the BMAs were made. Execution of the BMA gave reasonable results in terms of target assignment behavior.

For most simulation runs, a deployment of 2000 weapons-carrier platforms in a 64-ring Walker orbit constellation was used against a threat scenario of 1000 missiles launched essentially simultaneously from seven bases in an 80 degrees longitude arc through the Asiatic USSR.

In a successful effort to reduce the CPU time of the simulation, the KKV BMA helpers were modified to first test possible shots against search pruning criteria before calculating a trial KKV trajectory (a compute-intensive operation). The use of a conservative maxi-

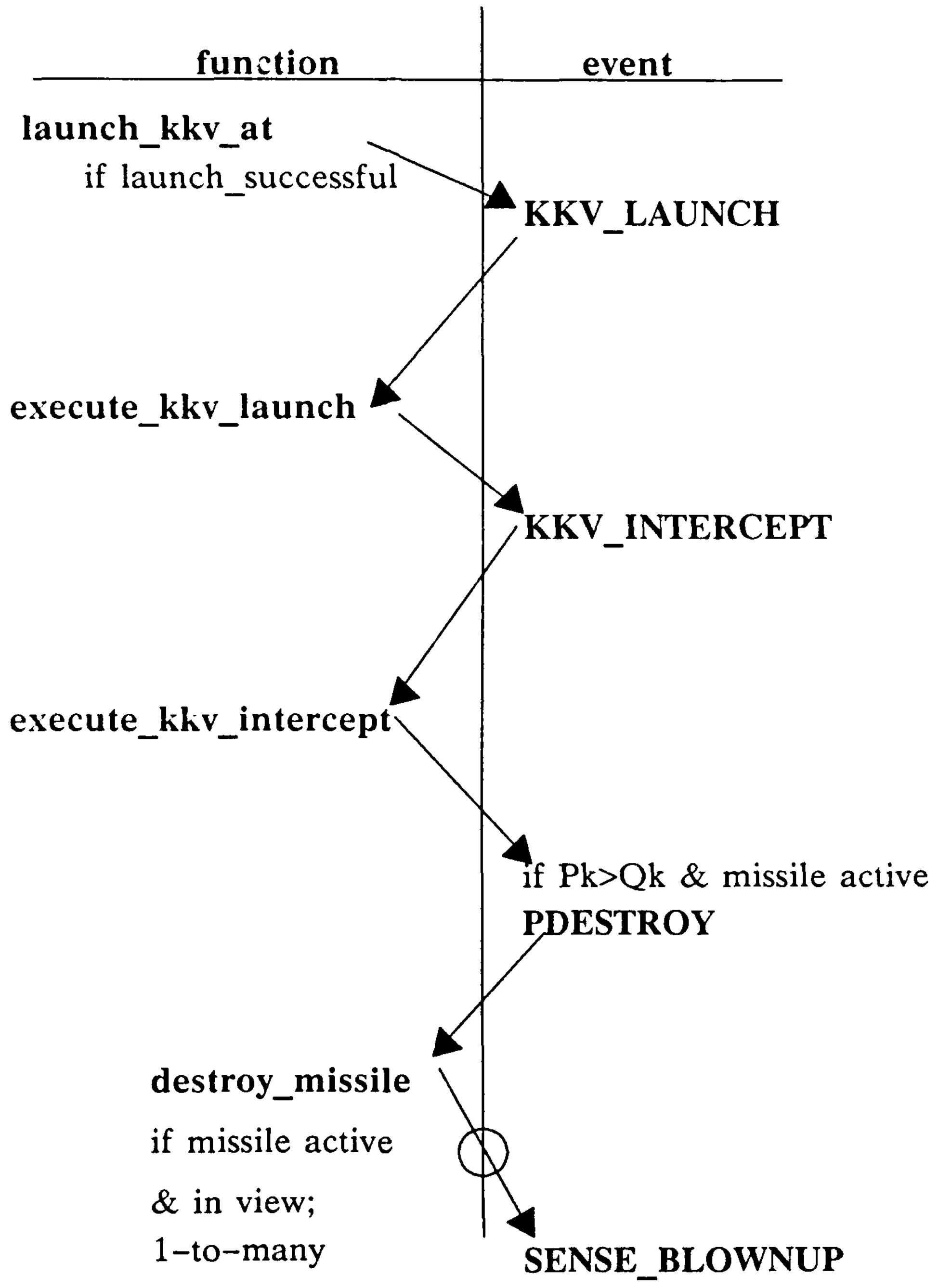

Figure 1. Events Related to KKVs

mum range heuristic reduced the search space by more than 97%. Better heuristics should make it possible to reduce the detailed search space to 1% of the total search space or less. With this heuristic efficiency, it is reasonable to allow the BMA to make a straightforward exhaustive search. This simplifies the BMA logic without incurring a prohibitive amount of execution time. The execution profile of the BMA further suggests that requests for positions and velocities of a given object tend to show a great deal of temporal locality. Therefore, a position/velocity cacheing scheme may provide further significant increases in performance.

These simulation runs (2000 platform deployment, 1000 missile threat) currently consume slightly more than one hour of CPU time on a VAX-11/8650 with a memory image of slightly more than 2MB. Position logging at 40-second [simulation time] intervals and at state change times produces output files of approximately 6MB. More than 90 percent of the CPU time is spent in calculating platform and missile positions and velocities (position/velocity cacheing should improve this tremendously), of which more than 95 percent are on behalf of the BMA helpers.

One interesting result obtained in testing the prototype BMA was that a deployment where the northernmost platform footprints were slightly to the south of the missile bases performed better (64 percent kill ratio compared to 54 percent, all other factors equal) than a deployment where some footprints passed north of the bases. The cause of this counterintuitive phenomenon was not determined, and may well be an artifact of the overly simplified simulation model; the significance lies in the fact that this result was obtained without any user coding, requiring only a few seconds' interaction to set up. The simulator may be a powerful tool to assist in searching for areas of unexpected SDI architecture performance, which could then be studied in greater depth.

6. Conclusions and Recommendations for Future Work

Basically, the experiment was a success. No significant difficulties were encountered in integrating the KKV technology module with the rest of the simulator. No "backtracking" of existing code was required. Strict changes to existing code, rather than additions to existing code, were limited to changing case statements, array sizes, etc. A more thoroughly table-driven design would remove even these small changes, and would make it possible to add technology modules without any code changes at all.

"Porting" the prototype BMA was extremely easy. In essence, the only required changes were replacing the laser weapon BMA helper calls with the equivalent KKV BMA helper calls, changing the BMA's idea of the weapon's "recharge" rate (the delay between launching a KKV and the launcher becoming ready again), and replacing the actual shoot_at() with launch_kkv_at(). The code structure did not change at all (some code changes were temporarily inserted to aid in debugging the KKV technology module).

The ease with which the prototype BMA was converted from laser weapons to KKVs prompts us to suggest the use, whenever feasible, of "generic" (i.e., not technology-specific) BMA interfaces in writing high-level simulations. The level of detail in the BMA interface should be consistent with the level of detail/fidelity in the underlying simulation model. Detailed knowledge about the expected behavior of a technology component should be restricted to the BMA helpers associated with that technology module. A reasonably simple and technology-independent BMA interface is still sufficient to support BMAs of fair sophistication.

The major problem encountered in the implementation of the KKV technology module occurred in kkv_opt_pk(), which attempts to find the launching time of a KKV from a given platform that would maximize Pk against a given target. In our KKV simulation model, this amounts to finding the launch time that has the shortest-duration flight to the intercept point. The iterative computation that we used to approximate the solution sometimes failed to converge. To the extent that future simulator users and developers with the requisite expertise in physics accept our time-of-flight criterion as a reasonable way to estimate Pk, those with the requisite expertise in mathematical and numerical analysis should revisit our solution to this nonlinear optimization problem.

Finally, we suggest that the appropriate next experiment to perform is the revision of an existing technology module, rather than the addition of a new one, as was done here. This would focus upon the issue discussed in Section 3 -- finding systematic ways of increasing the amount of detail in the interface between a BMA and a technology module, when that technology module is replaced with a higher-fidelity one.

References

[1] Mizell, D., Tung, Y., Coatney, S., Carter, S., Sherman, R., and Najjar, W., "On the Design of ISI's Initial Prototype SDI Architecture Simulator," ISI Research Report, ISI/RR-88-205, March 1988.

[2] Mizell, D. and Carter, S., "ISI's SDI Architecture Simulator: The 'Kmac' Battle Manager Specification Language," ISI Research Report, in preparation.

Appendix: Summary of Software Development Necessary to Implement the KKV Technology Module

Functions required in the KKV technology module:

1) Code to model the behavior of the KKV launcher

2) Code to create a KKV trajectory given a platform, a target, and a launch time

3) Code to calculate the position and velocity of an active KKV as a function of time

4) Code to write positions of all active KKVs to the display database

5) BMA helpers to predict Pk for various possible shots and to find optimal KKV launch times

Code structure of the new technology module:

The KKV technology module is contained in the new files kkv.c and kkv.h.
The technology module contains the following C++ classes:

1) class kkv - the basic KKV object
2) class kkv_list - a list of all KKVs in the system analogous to the classes platform_list and missile list
3) class kkv_list_iterator - a class which allows a function to step through the entire set of kkvs while hiding the representation of the kkv_list
4) class kkv_launcher - this class is the weapon capability, conceptually included in the platform which has the capability

The technology module also contains the following interface functions whose operation has been previously described:

1) BMA interface functions shoot_kkv_at(), kkv_pk(), kkv_opt_pk()
2) Simulator interface functions execute_kkv_launch() and execute_kkv_intercept()

MODIFICATIONS TO EXISTING FILES

The following is intended as a guide to a programmer making enhancements to the simulator and assumes familiarity with the general structure of the simulator and the files that comprise it.

I) Add the new capability [KKV_LAUNCHER]

1) Modify global.h to include the new weapon type and change NUM_WEAPON_TYPES; add the string to weapon_type strings[] in global.c and xweapon_type_strings[] in make_capabilities.c

2) Add capability_descriptor::get_kkv_launcher() and (which is actually the same function) platform::get_kkv_launcher() to objects.h and objects.c and modify

II) Add the new event types [KKV_LAUNCH and KKV_INTERCEPT]

1) Add the new event types added to the enum event_type in global.h

2) Add the declaration of new event types' data content (kkv_launch_struct and kkv_intercept_struct) and the new event constructors to event.h and event.c. Add formatted prints (operator<<) of the new event types and function event::get_kkv_ptr() to event.c

3) Add the code to execute the events (basically calls to execute_kkv_launch() and execute_kkv_intercept() in kkv.c) to simulator.c

4) Add the new events to make_flags.c

III) add the new object type [kkv] The existence of the KKV objects is essentially unknown to the rest of the simulator, e.g., sensors do not report the presence of KKVs. The position of active KKVs is logged to the position database for subsequent display [and/or analysis using tools not currently developed]. This required additions to print_objects() in simulator.c

IV) add the new trajectory type [TYP_KKV]

Trajectories were implemented as a single class that contained a union of the various subtypes [missile trajectory, different orbit types, etc.]. Each member function of the trajectory class (e.g., traj::pos_and_vel(), which returns the position and velocity of an object at a given time) contains a switch statement over the different allowed trajectory types. All these switch statements in traj.c were expanded to call the appropriate routine from kkv.c when called with a KKV trajectory object.

This opportunity was taken to expand the general trajectory package with some general purpose functions, notably functions to create and apply direction cosine matrices for inertial to earth-fixed coordinate systems.